Chap. 6

Architectural Support for High-Level Languages

6.1 Abstraction in software design

- Assemble-level abstraction
  - A programmer who writes directly with the raw machine instruction set
  - Expressing the program in terms of instructions, addresses, registers, bytes and words
- High-level languages
  - Allows the programmer to think in terms of abstractions that are above the machine level
  - The programmer may not even know on which machine the program will ultimately run
  - The RISC philosophy focusing instruction set design on flexible primitive operations from which the complier can build its high-level operations
- This chapter
  - describes the requirements of high-level languages and shows how they are met by the ARM architecture

6.2 Data types

- ARM support for characters
  - For handling characters is the unsigned byte load and store instruction
- ANSI (American National Standards Institute) C basic data types
  - Defines the following basic data types
    - Signed and unsigned characters of at least eight bits
    - Signed and unsigned short integers of at least 16 bits
    - Signed and unsigned integers of at least 16 bits
    - Signed and unsigned long integers of at least 32 bits
    - Floating-point, double and long double floating-point numbers
    - Enumerated types
    - Bitfields (sets of Boolean variables)
  - The ARM C complier adopts the minimum sizes for each of these types
  - The standard integer uses 32-bit values

6.2 Data types

- ANCI C derived data types
  - Defines derived data types
    - Arrays, Functions, Structures, Pointers, Unions
  - ARM pointers are 32 bits long and resemble unsigned integers
  - The ARM C compiler aligns characters on byte boundaries, short integers at even addresses and all other types on word boundaries
- ARM architectural support for C data types
  - Provides native support for signed and unsigned 32-bit integers and for unsigned bytes, covering the C integer, long integer and unsigned character types
  - For arrays and structures: base plus scaled index addressing Chap. 5, p.38
  - Current versions of the ARM include signed byte and signed and unsigned 16-bit loads and stores, providing some native support for short integer and signed character types

6.3 Floating-point data types

- Floating-point number
  - Attempt to represent real numbers with uniform accuracy

$$R = a \times b^n$$

    - n is chosen so that a falls within a defined range of values
    - b is usually implicit in the data type and is often equal to 2

- IEEE 754
  - There are many complex issues to resolve with the handling of floating-point numbers in computers to ensure that the results are consistent when the same program is run on different machines
  - The consistency problem was greatly aided by the introduction in 1985 of the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standard 754-1985, sometimes referred to simply as IEEE 754)

6.3 Floating-point data types

- Single precision
  - 32-bit format
    - S: sign bit
    - exponent: an unsigned integer value with a 'bias' of +127
    - fractional

$$\text{value (norm)} = (-1)^S \times 1.\text{fraction} \times 2^{(\text{exponent}-127)}$$

- Double precision
  - 64-bit format

$$\text{value (norm)} = (-1)^S \times 1.\text{fraction} \times 2^{(\text{exponent}-1023)}$$

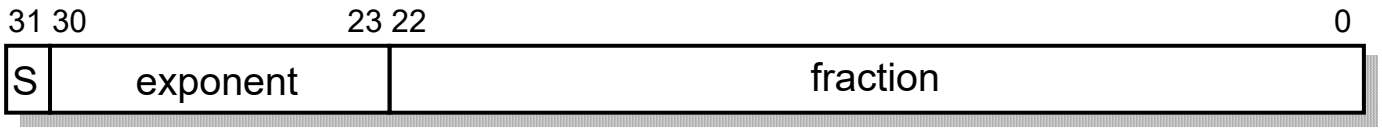


Fig. 6.1 IEEE 754 single precision floating-point number format

easily processed by integer comparisons, especially for sorting

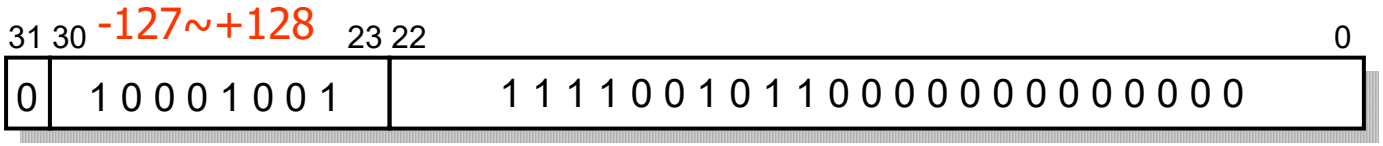


Fig. 6.2 IEEE 754 single precision representation of '1995'

$$\text{value (norm)} = (-1)^{S} \times 1.\text{fraction} \times 2^{(\text{exponent}-127)}, \quad (0\sim255)$$

$$1995 = 11111001011_2 = 1.1111001011 \times 2^{1010}$$

exponent−127=10, exponent =137=10001001

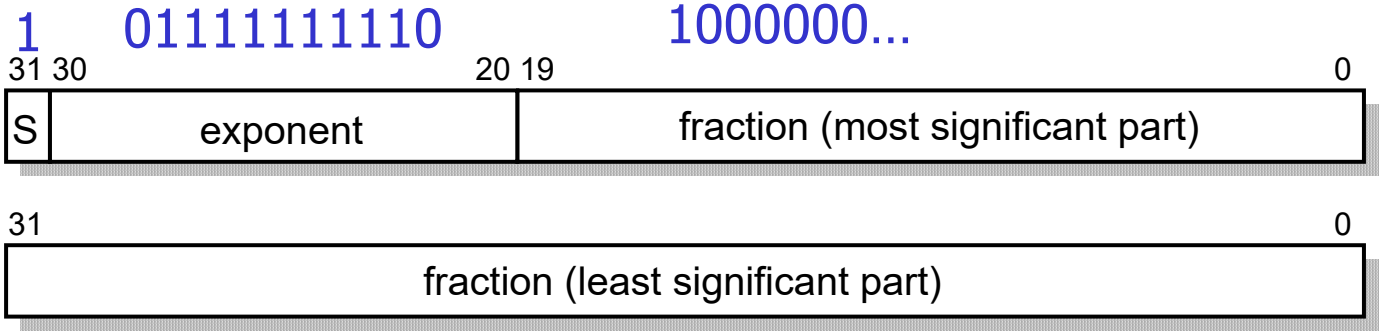


Fig. 6.3 IEEE 754 double precision floating-point number format

$$\text{value (norm)} = (-1)^S \times 1.\text{fraction} \times 2^{(\text{exponent}-1023)}$$

$-0.75 = -0.11_2 = -1.1 \times 2^{-1}$

exponent$-1023 = -1$, exponent $= 1022 = 01111111110$

Special number representation

- Zero: zero exponent and fraction with either sign value.
- Infinity: maximum exponent value with a zero fraction and the appropriate sign bit.
- NaN: maximum exponent and a non-zero fraction.
- Denormalized number: zero exponent and non-zero fraction.

$$\text{value (denorm)} = (-1)^{S} \times 0.\text{fraction} \times 2^{(-126)}$$

6.3 Floating-point data types

- Other Formats
  - Double extended precision (80 bits), Packed decimal, Extended packed decimal
- ARM floating-point instructions
  - ARM limited has defined a set of floating point instructions within the coprocessor instruction space
  - Implemented entirely in software through the undefined instruction trap
  - A subset may be handled in hardware by the FPA10 floating-point coprocessor
- ARM floating-point library
  - ARM Limited also supplies a C floating-point library which supports IEEE single and double precision formats

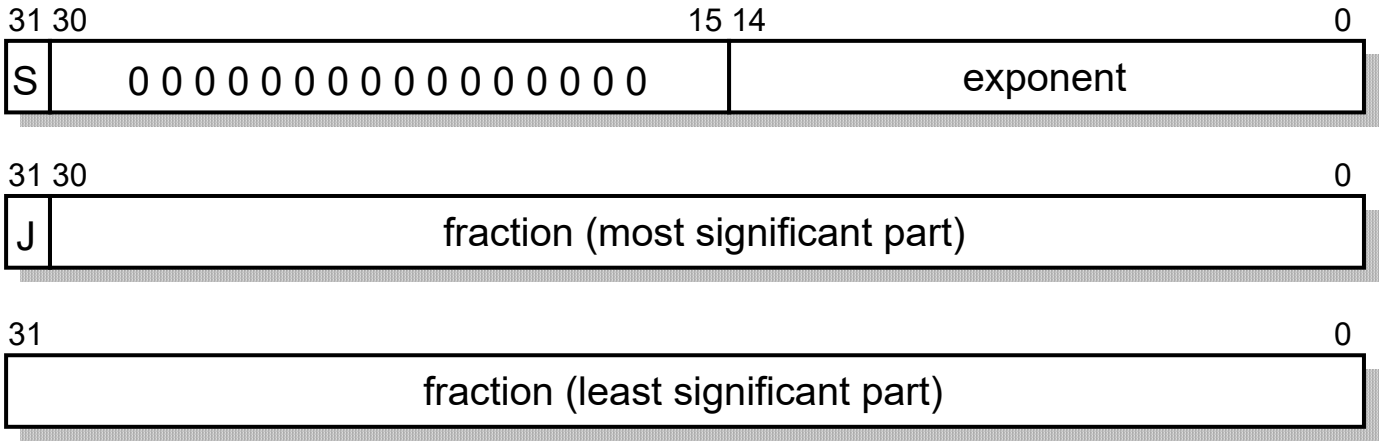


Fig. 6.4 IEEE 754 double extended precision floating-point number format

$$\text{value (packed)} = (-1)^{D} \times \text{decimal} \times 10^{((-1)^{E} \times \text{exponent})}$$

$$1 \le \text{decimal} < 10$$

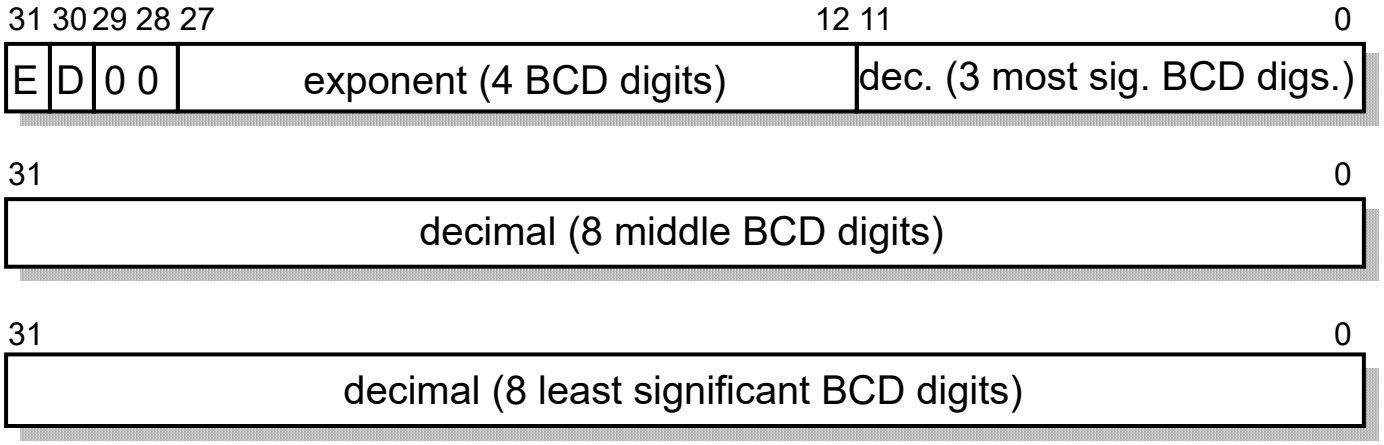


Fig. 6.5 IEEE 754 packed decimal floating-point number format

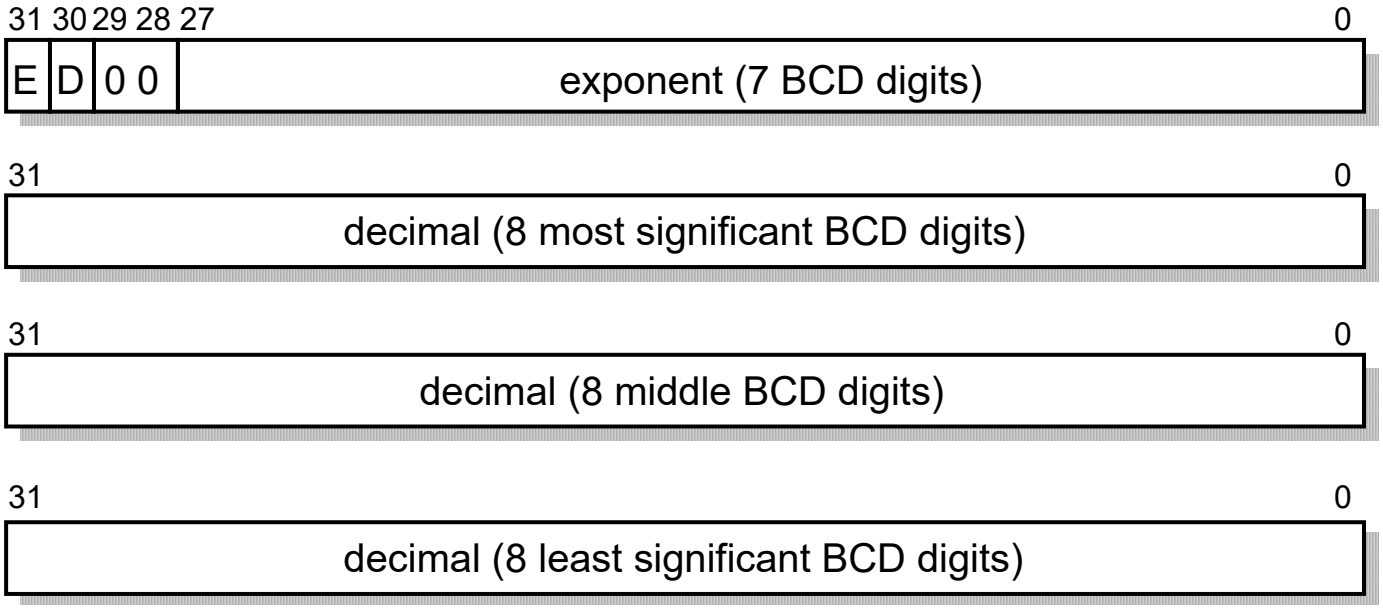


Fig. 6.6 IEEE 754 extended packed decimal floating-point number format

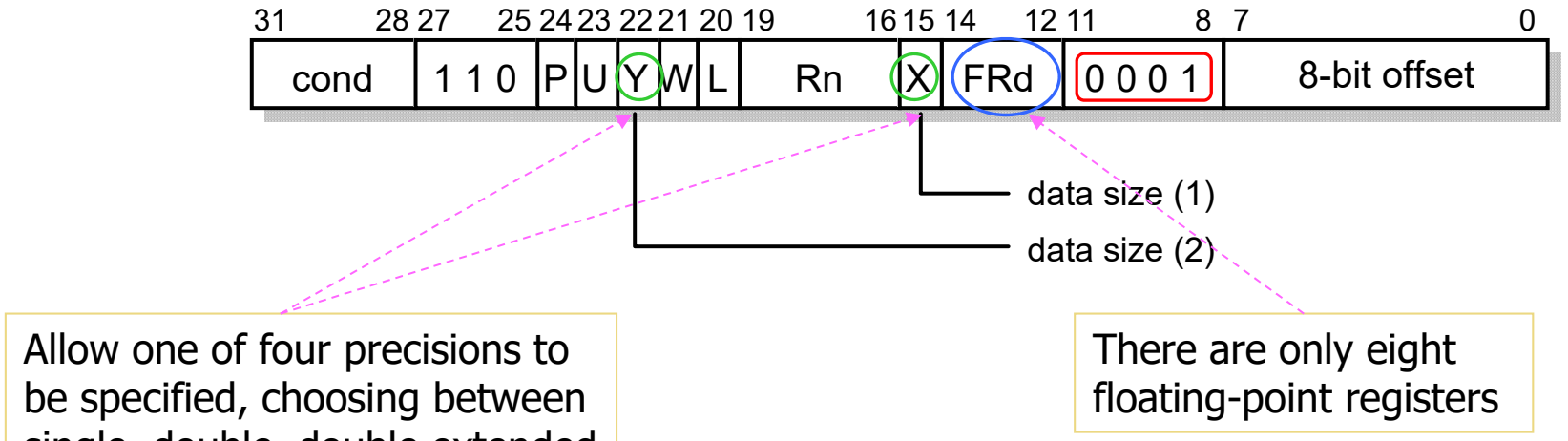
6.4 The ARM floating-point architecture

- The ARM floating-point architecture provides extensive support
  - Either entirely in software
  - Or using a combined software/hardware solution based around the FPA10 floating-point accelerator
- The ARM floating-point architecture presents
  - An interpretation of the coprocessor instruction set when the coprocessor number is 1 or 2
  - Eight 80-bit floating-point registers in coprocessors 1 and 2
  - A user-visible floating-point status register (FPSR) which controls various operating options and indicates error conditions
  - Optionally, a floating-point control register (FPCR) which is user-invisible

6.4 The ARM floating-point architecture

- FPA10 data types
  - Supports single, double, and extended double precision formats
  - The coprocessor registers are all extended double precision
  - Loads and stores between memory and these registers can convert the precision as required

- Load and store floating instructions



6.4 The ARM floating-point architecture

- Load and store floating instructions

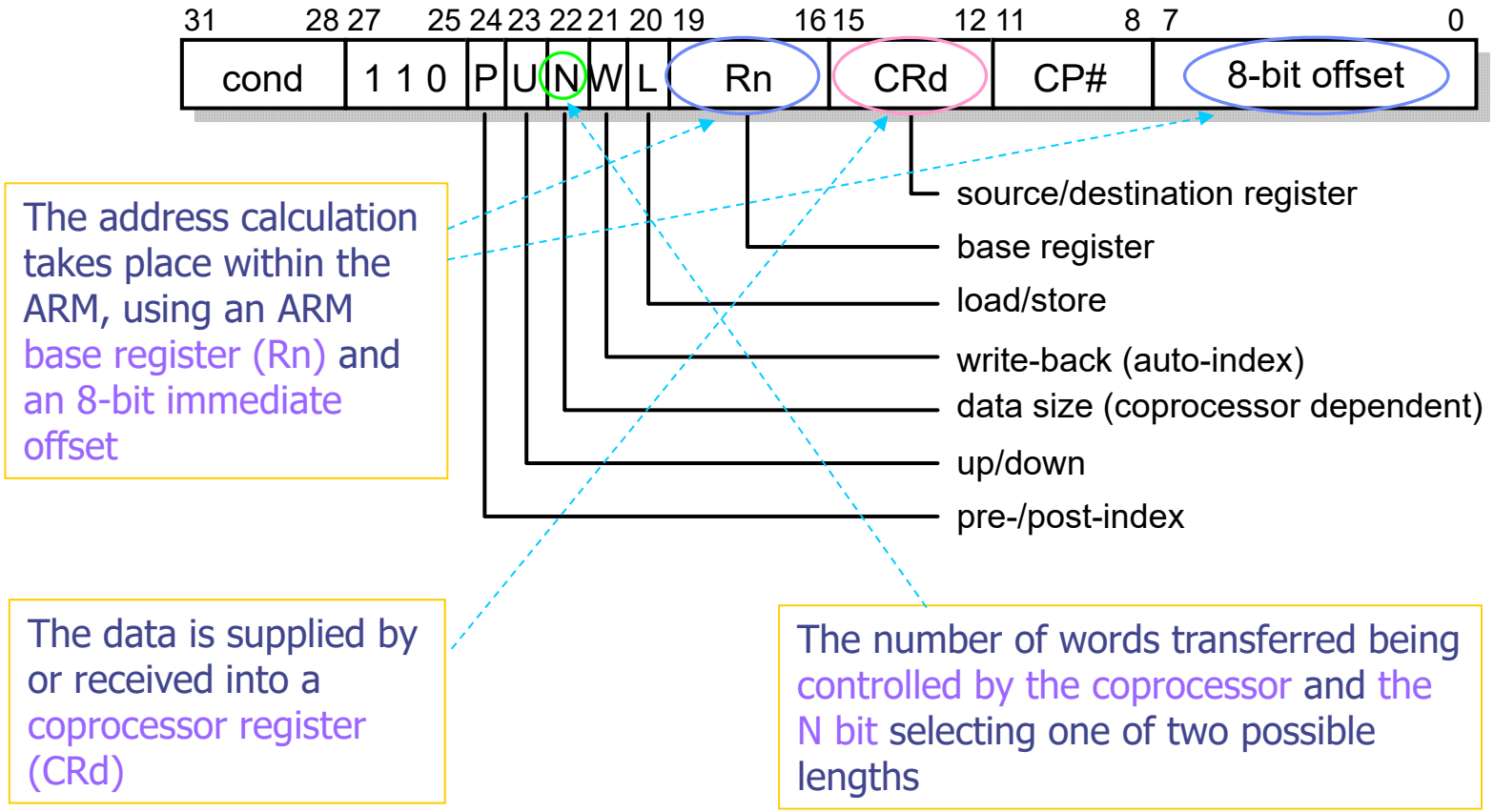
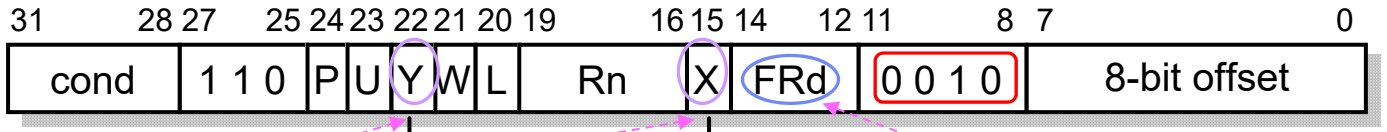


Fig. 5.16 Coprocessor data transfer instruction binary encoding

6.4 The ARM floating-point architecture

- Load and store multiple floating
  - Used to save and restore the floating-point register state
  - Each register is saved using three memory words



Encode the number of registers transferred which can be from one to four

register count (1)

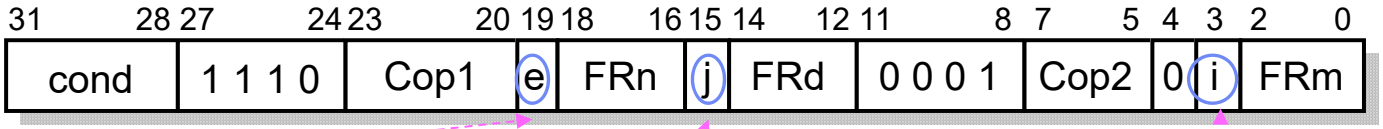
register count (2)

Specifies the first register to be transferred

These instructions use coprocessor number 2, whereas the other floating-point instructions use coprocessor number 1

6.4 The ARM floating-point architecture

- Floating-point data operations
  - Perform arithmetic functions on values in the floating-point register
  - Has a number of opcode bits, augmented by extra bits from each of the three register specifier field since only three bits are required to specify one of the eight floating-point
  - The instructions include simple arithmetic operations (add, subtract, multiply, divide, remainder, power), transcendental functions (log, exponential, sin, cos, tan, ….) and assorted others (square root, move…..)



'e' and 'Cop2' control the destination size and the rounding mode

Select between single operand and two operand operations

Selects between a register ('FRm') or one of eight constants for the second operand

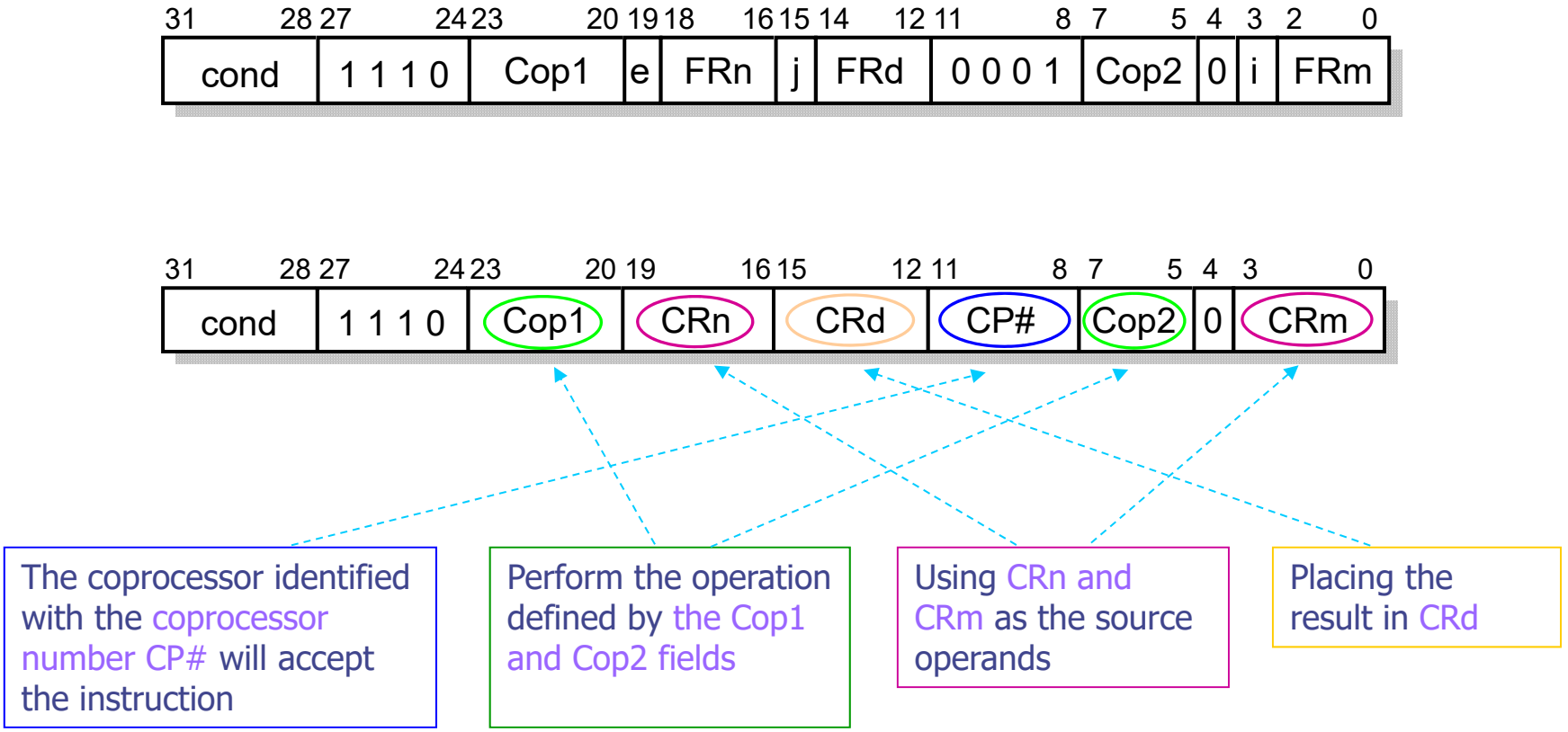
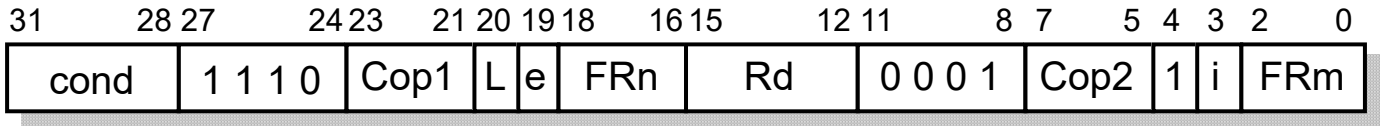


Fig. 5.15 Coprocessor data processing instruction binary encoding

- Examples
  - CDP p2, 3, C0, C1, C2
  - CDPEQ p3, 6, C1, C5, C7, 4

6.4 The ARM floating-point architecture

- Floating-point register transfers
  - accept a value from or return a value to an ARM register
  - From ARM to the floating-point unit include
    - – ‘float’: convert an integer in an ARM register to a real in a floating-point register, and
    - – writes to the floating-point status and controls
  - From the floating-point unit to ARM
    - – ‘fix’: convert a real in a floating-point register to an integer in an ARM register, and
    - – reads of the status and control registers



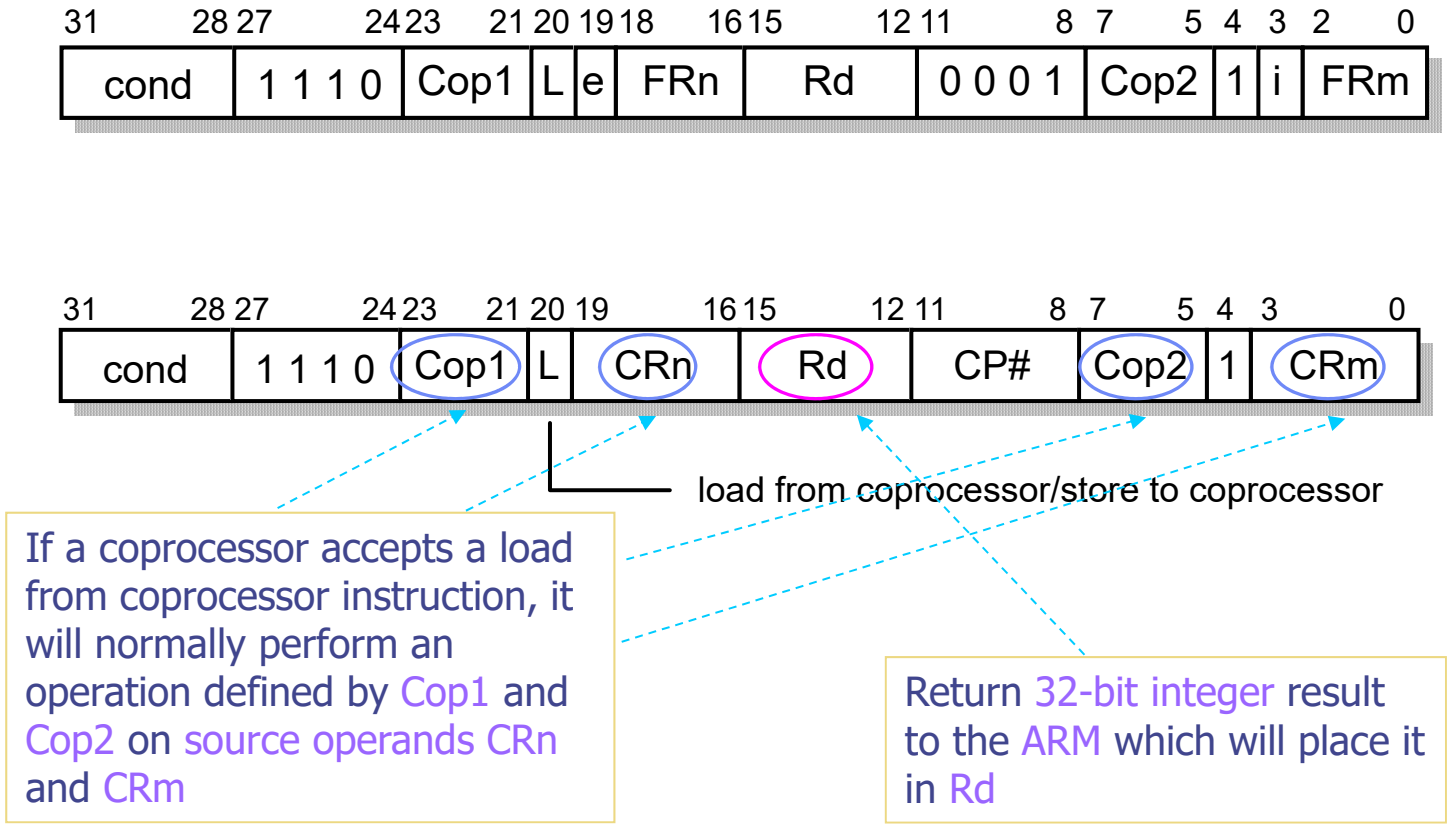


Fig. 5.17 Coprocessor register transfer instruction binary encoding

6.4 The ARM floating-point architecture

- Assembler format
  - Move to ARM register from coprocessor

    MRC{<cond>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}
  - Move to coprocessor from ARM register

    MCR{<cond>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}
- Examples
  - MCR p14, 3, r0, C1, C2
  - MRCCS p2, 4, r3, C3, C4, 6
- Notes
  - The coprocessor must perform some internal work to prepare a 32-bit value for transfer to the ARM, it will often be necessary for the coprocessor handshake to 'busy-wait' while the data is prepared
  - Transfers from the ARM to the coprocessor are generally simpler since any data conversion work can take place in the coprocessor after the transfer has completed

6.4 The ARM floating-point architecture

- Floating-point instruction frequencies

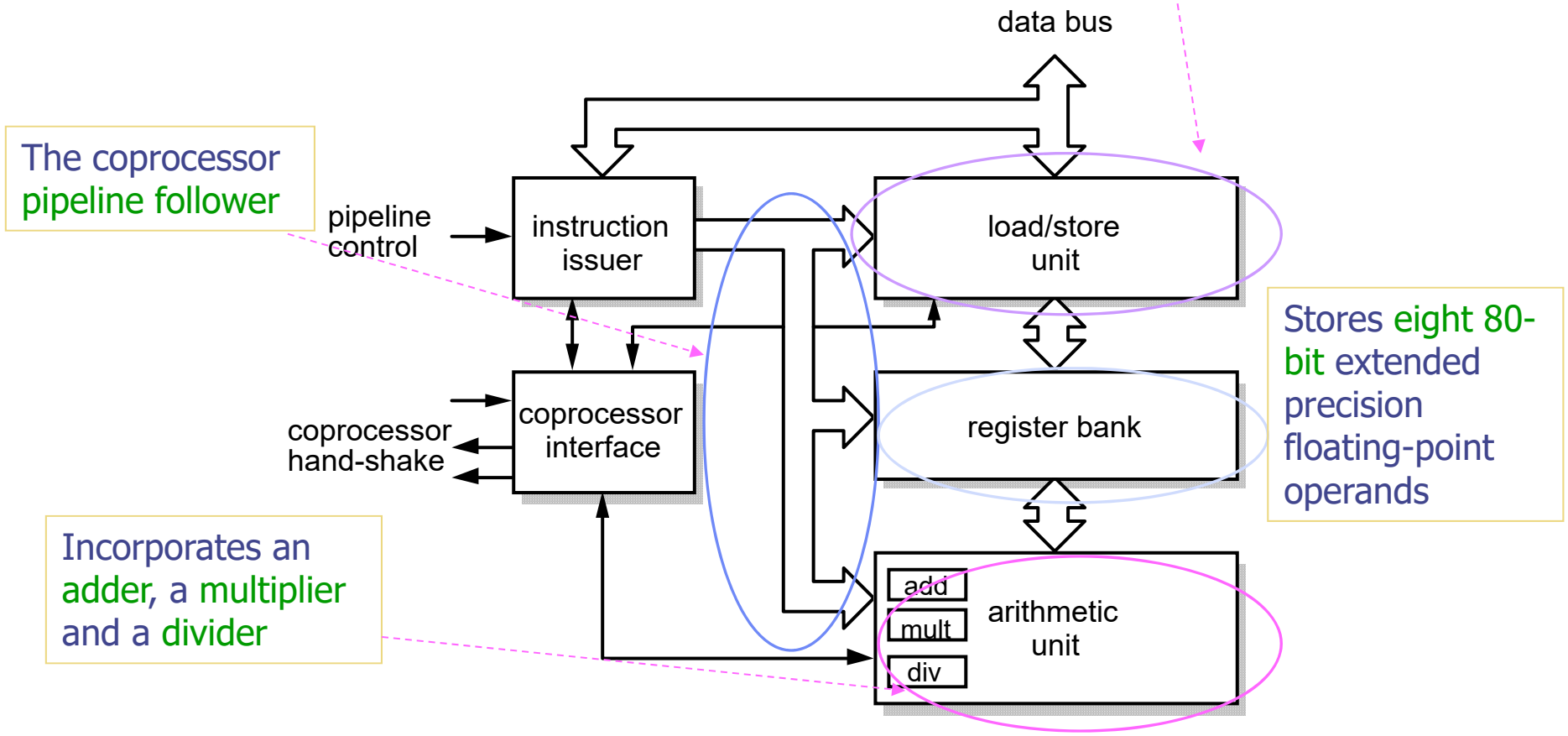
Instruction	Frequency
Load/store	67%
Add	13%
Multiply	10.5%
Compare	3%
Fix and float	2%
Divide	1.5%
Others	3%

Floating-point instruction frequencies

6.4 The ARM floating-point architecture

- FPA10 organization

Carries out format conversion on floating-point data types as they are loaded from and stored to memory



6.4 The ARM floating-point architecture

- FPA10 pipeline
  - The FPA10 arithmetic unit operates in four pipeline stages:
    - Prepare: align operands
    - Calculate: add, multiply or divide
    - Align:normalize the result
    - Round: apply appropriate rounding to the result
- FPA10 applications
  - is used as a macrocell on the ARM7500FE chip
- The VFP10
  - a much higher performance floating-point unit, the VFP10, has been designed to operate with the ARM10TDMI processor core

6.5 Expressions

- Register use
  - The key to the efficient evaluation of a complex expression is to get the required values into the registers in the right order and to ensure that frequently used values are normally resident in registers
  - Optimizing this trade-off between the number of values that can be held in registers and the number of registers remaining is a major task for the complier
- ARM support
  - The 3-address instruction format used by the ARM gives the compiler the maximum flexibility
  - Thumb instructions (generally 2-address)
    - restricts the compiler's freedom to some extent
    - smaller number of general registers also makes its job harder

6.5 Expressions

- Accessing operands
  - A procedure will normally work with operands that are presented in one of the following ways, and can be accessed as indicated
  - As an argument passed through a register
    - The value is already in a register, so no further work is necessary
  - As a argument passed on the stack
    - Stack pointer (r13) relative addressing with an immediate offset known at compile-time allows the operand to be collected with a single LDR
  - As a constant in the procedure's literal pool
    - PC-relative addressing, again with an immediate offset known at compile-time, gives access with a single LDR
  - As a local variable
    - Local variables are allocated space on the stack and are accessed by a stack pointer relative LDR
  - As a global variable
    - Global (and static) variables are allocated space in the static area and are accessed by static base (is usually in r9) relative addressing

6.5 Expressions

- Pointer arithmetic
  - Arithmetic on pointers depends on the size of the data types that the pointers are pointing to

```
int *p;
p = p+1;
```
⇒ increase the value of p by 4 bytes

  - If a variable is used as an offset it must be scaled at run-time

```
int i=4;
p = p+i;
```

if p is held in r0 and i in r1, the change to p may be compiled as:

```
ADD    r0, r0, r1, LSL #2    ; scale r1 to int
```

- Arrays
  - The declaration: int a[10];
    - a reference to a[i] is equivalent to the pointer-plus-offset form *(a+i)

6.6 Conditional statements

- if…else
  - The ARM architecture offers unusually efficient support for conditional expressions when the conditionally executed statement is small

```
          r0  r1  r2
 if (a>b) c=a; else c=b;
```

```
CMP     r0, r1          ; if (a>b)...
MOVGT   r2, r0          ;  ..c=a..
MOVLE   r2, r1          ; ...else c=b
```

  - More complex ‘if’ statements: more conventional solution

```
        CMP     r0, r1          ; if (a>b)...
        BLE     ELSE            ; skip clause if false
        MOV     r2, r0          ;  ..c=a..
        B       ENDIF           ; skip else clause
ELSE    MOV     r2, r1          ; ...else c=b
ENDIF   ..
```

6.6 Conditional statements

- switches
  - ARM is to use a jump table

```
switch (expression) {
  case constant-expression1: statements1
  case constant-expression2: statements2
  ...
  case constant-expressionN: statementsN
  default: statementsD
}
```

```
temp = expression;
if (temp==constant-expression1) {statements1}
else ...
else if (temp==constant-expressionN) {statementsN}
else {statementsD}
```

This can result in slow code if the switch statement has many cases

6.6 Conditional statements

```
switch (expression) {
  case constant-expression1: statements1
  case constant-expression2: statements2
  ...
  case constant-expressionN: statementsN
  default: statementsD
}
```

```
JUMPTABL    B   L1   ; r0=0
            B   L2   ; r0=1
            ...
            B   LN   ; r0=N-1
```

```
                ; r0 contains value of expression
        ADR     r1, JUMPTABLE       ; get base of jump table
        CMP     r0, #TABLEMAX       ; check for overrun..
        LDRLS   pc, [r1,r0,LSL #2]  ;  .. if OK get pc
                ; statementsD       ;  .. otherwise default
        B       EXIT                ; break
L1      ..      ; statements1
        B       EXIT                ; break
        ..
LN      ..      ; statementsN
EXIT    ..
```

6.7 Loops

- For loops
  - for (i=0; i<10; i++) {a[i] = 0; } (p173)

r0

```
        MOV     r1, #0              ; value to store in a[i]
        ADR     r2, a[0]            ; r2 points to a[0]
        MOV     r0, #0              ; i=0
LOOP    CMP     r0, #10             ; i<10 ?
        BGE     EXIT                ; if i >= 10 finish
        STR     r1, [r2,r0,LSL #2]; a[i] = 0
        ADD     r0, r0, #1          ; i++
        B       LOOP
EXIT    ..
```

0

  - It can be further improved
    - by omitting 'BGE EXIT' and applying the opposition condition to the following instructions
    - or, by moving the test to the bottom of the loop

6.7 Loops

- While loops
  - The standard conceptual arrangement of a ‘while’ loop is as follow

```
LOOP    ..                  ; evaluate expression
        BEQ     EXIT
        ..                  ; loop body
        B       LOOP
EXIT    ..
```

  - One fewer branch is executed each time the complete ‘while’ structure is encountered

```
        ..                  ; evaluate expression
        BEQ     EXIT        ; skip loop if necessary
LOOP    ..                  ; loop body
TEST    ..                  ; evaluate expression
        BNE     LOOP
EXIT    ..
```

6.8 Functions and procedures

- Program design
  - Large programs are broken down into components that are small enough be thoroughly tested
  - How small software component performs its operation should be of no significance to the rest of the program (abstraction)
- Program hierarchy
  - The full program should be designed as a hierarchy of components
  - Lower-level routines may be shared by higher-level routines, calls may skip levels, and the depth may vary across the hierarchy
- Leaf routines
  - At the lowest level of the hierarchy
  - Do not call any lower-level routines
  - Bottom-level routines will be library or system functions

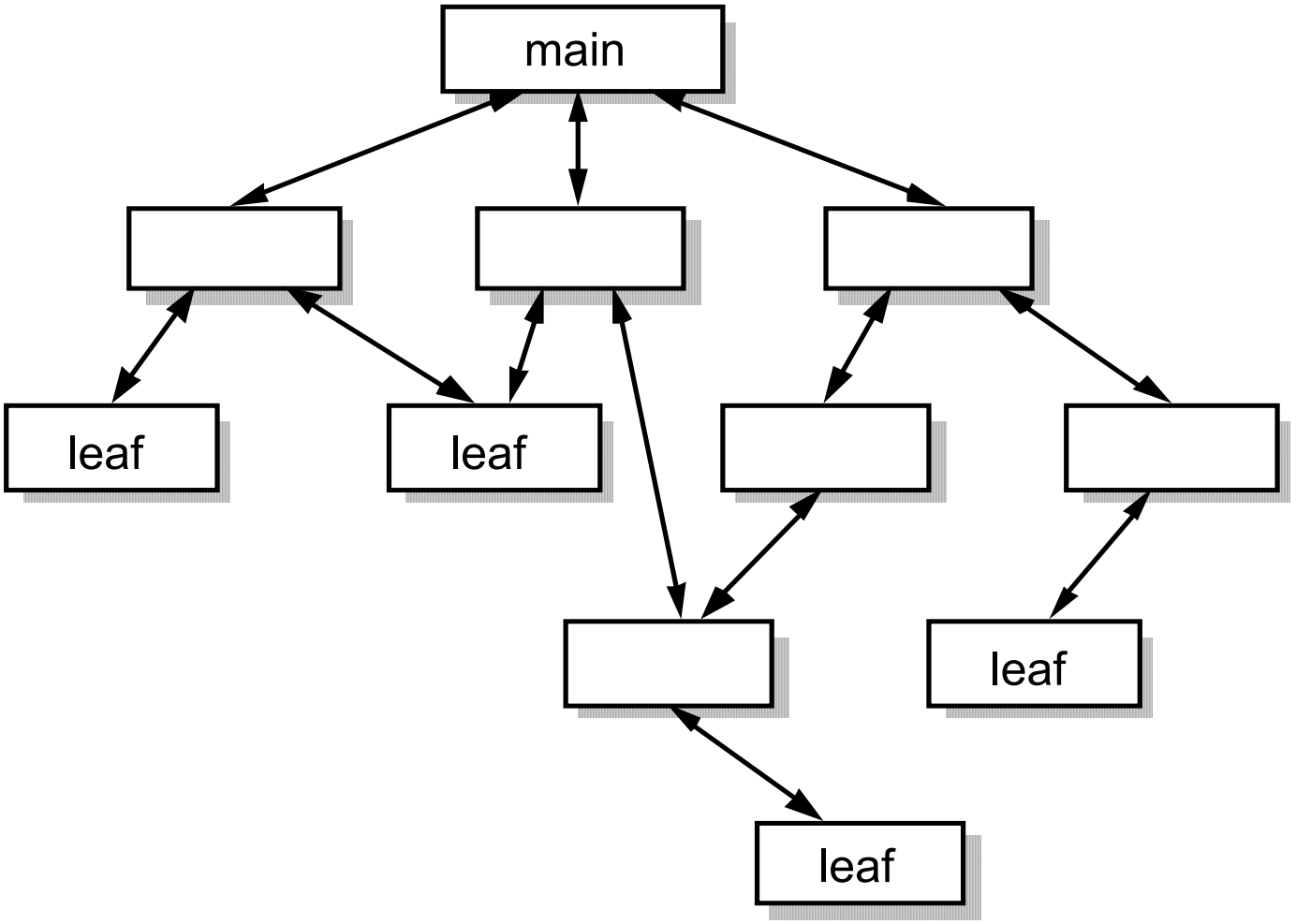


Fig. 6.12 Typical hierarchical program structure

6.8 Functions and procedures

- Terminology
  - Subroutine
    - A generic term for a routine that is called by a higher-level routine, particularly when viewing a program at the assembly language level
  - Function
    - A subroutine which returns a value through its name

```
c = max (a, b);
```

  - Procedure
    - A subroutine which is called to carry out some operation on specified data item(s)

```
printf ("Hello World\n");
```

  - In C, all subroutines are functions.
  - Argument: an expression passed to a function call
  - Parameter: a value received by the function
  - C uses a strict "call by value" semantics instead of "call by reference".

6.8 Functions and procedures

- ARM Procedure Call Standard
  - The ARM Procedure Call Standard (APCS) is employed by the ARM C compiler
  - It defines particular uses for the 'general-purpose' registers
  - It defines which form of stack is used from the full/empty, ascending/descending choices
  - It defines the format of a stack-based data structure used for back-tracing when debugging programs
  - It defines the function argument and result passing mechanism to be used by all externally visible functions and procedures
  - It supports the ARM shared library mechanism
    - which means it supports a standard way for shared (re-entrant) code to access static data

6.8 Functions and procedures

- APCS register usage
  - Table 6.2: the 16 visible ARM registers are divided into three sets:
    - Four argument registers which pass values into the function
      - The function need not preserve these so it can use them as scratch registers once it has used or saved its parameter values
      - They must be saved across such calls if they contain values that are need again ⇒ They are caller-saved register variables
    - Five (to seven) register variables which the function must return with unchanged values
      - These are callee-saved register variable
      - This function must save them if it wishes to use the registers
    - Seven (to five) registers which have a dedicated role
      - The link register (lr), for example, carries the return address on function entry, but if it is saved (the function calls subfunctions) it may then be used as a scratch register

```
int func1 (int a);
void func2 (int b);


int main()
{
                int a, b;
                int x[10];

                b = 7;
                x[2]=28;

                a=func1(x[2])
                func2(a+b);

        return 0;
}

int func1 (int a)
{
        int y[5], b;

        b = y[1]+a;
        func2 (b);
        return (b);
}

void func2 (int b)
{
}
```

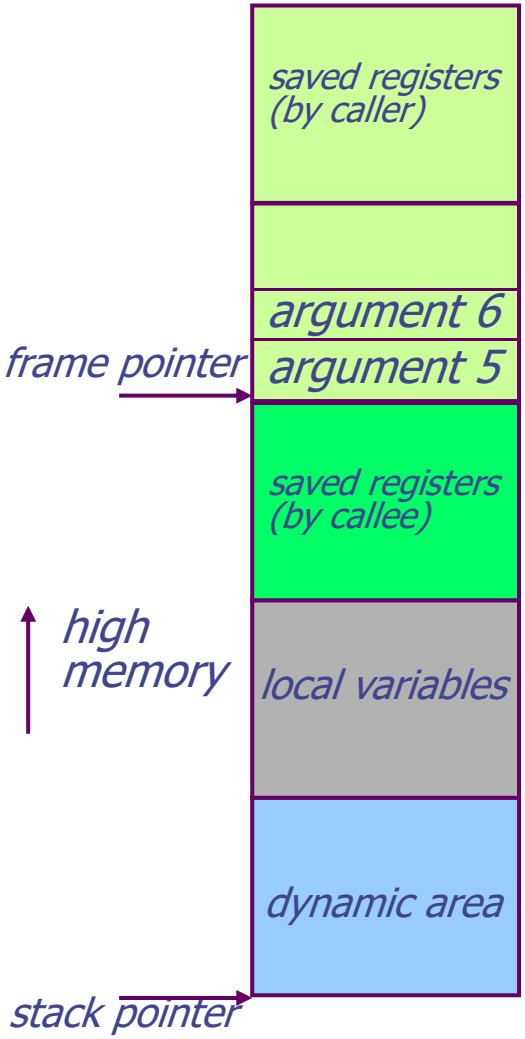
```
func2
        0x00000000 MOV      pc,r14
func1
        0x00000004 STR      r14,[r13,#-4]!
        0x00000008 RSB      r1,r0,r0,LSL #3
        0x0000000c MOV      r0,r1
        0x00000010 BL       func2   ; 0x0
        0x00000014 MOV      r0,r1
        0x00000018 LDR      pc,[r13],#4
main
        0x0000001c STR      r14,[r13,#-4]!
        0x00000020 SUB      r13,r13,#0x28
        0x00000024 MOV      r0,#0x1c
        0x00000028 STR      r0,[r13,#8]
        0x0000002c BL       func1   ; 0x4
        0x00000030 ADD      r0,r0,#7
        0x00000034 BL       func2   ; 0x0
        0x00000038 MOV      r0,#0
        0x0000003c ADD      r13,r13,#0x28
        0x00000040 LDR      pc,[r13],#4
```

Table 6. 2 APCS register use convention

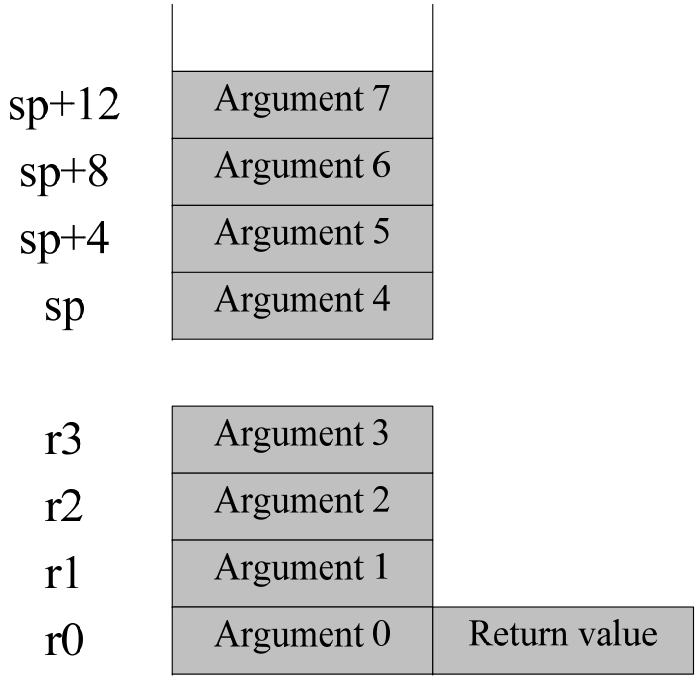
Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

Calling Convention

- GCC calling convention
  - first 4 arguments saved in registers
  - registers saved by caller and callee (including frame pointer and returning PC)
  - frame pointer points just below the last argument passed on the stack (the bottom of the frame)
  - stack pointer points to the first word after the frame



ATPCS argument passing



6.8 Functions and procedures

- APCS variants
  - There are several (16) different variants of the APCS which are used to generate code for a range of different systems, they support:
  - 32- or 26- bit PCs
    - Older ARM processors operated in a 26-bit address space
    - Some later versions continue to support this for backwards compatibility
  - Implicit or explicit stack-limit checking
    - The compiler can insert instructions to perform explicit checks for stack overflow
    - Where memory management hardware is available, an ARM system can allocate memory to the stack in units of a page (no need to insert instructions)
  - Two way to pass floating-point arguments
    - Make extensive use – use eight floating-point registers
    - Make little or no use – use the integer registers and/or on the stack
  - Re-entrant or non-re-entrant code
    - Code as re-entrant is position-independent and addresses all data indirectly through the static base register (sb)
    - This code can be placed in a ROM and can be shared by several processes

6.8 Functions and procedures

- Result return
  - A simple result is returned through a1
  - A more complex result is returned in memory

- Function entry and exit
  - A simple leaf function
    - which can perform all its functions using only a1 to a4 can be compiled into code with a minimal calling overhead
  - Where registers must be saved
    - the function must create a stack frame
    - this can be compiled efficiently using ARM's load and store multiple instructions

```
        BL    leaf1
        ..
leaf1   ..
        MOV   pc, lr   ; return
```

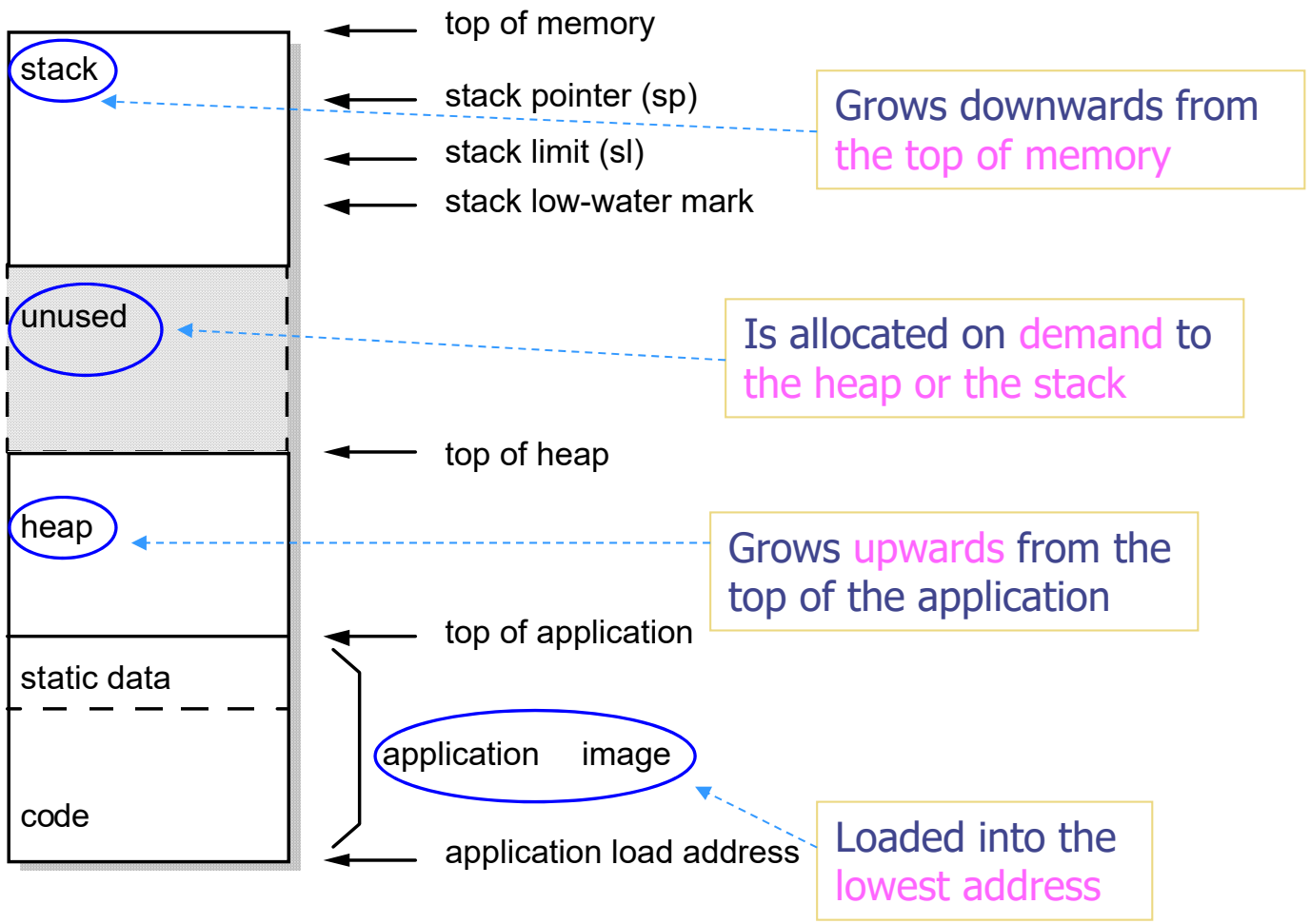
```
        BL      leaf2
        ..
leaf2   STMFD   sp!, {regs, lr}    ; save registers
        ..
        LDMFD   sp!, {regs, pc}    ; restore and return
```

6.9 Use of memory

- A C program expects to have access to
  - A fixed area of program memory
  - Memory to support two data areas that grow dynamically
    - The compiler often cannot work out a maximum size
- These dynamic data areas are
  - The stack
    - Whenever a function is called, a new activation frame is created on the stack containing a backtrace record, local variables, and so on
  - The heap
    - The heap is an area of memory used to satisfy program requests (malloc()) more memory for new data structures

6.9 Use of memory

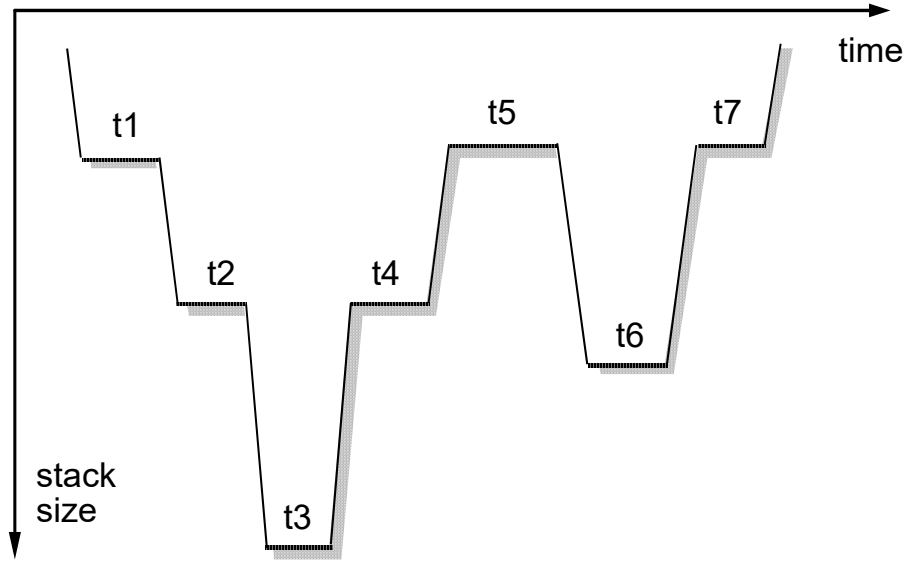
- Address space model



6.9 Use of memory

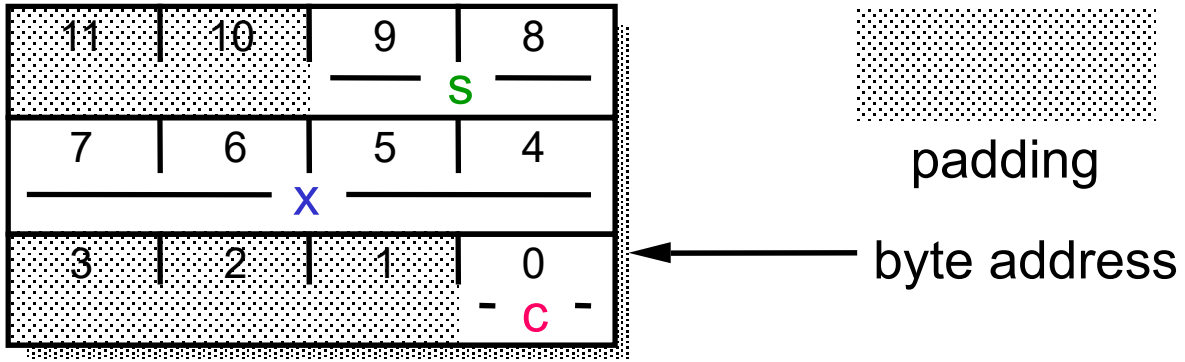
- Stack behaviour
  - Assuming that the compiler allocates stack space for each function call, the stack behaviour will be as shown in Fig. 6.14
  - At each function call, stack space is allocate
    - for arguments (if they cannot all be passed in registers)
    - to save registers for use within the function
    - to save the return address and the old stack pointer
    - to allocate memory on the stack for local variables

```
main () {
  ..                /* t1 */
  func1 ();
  ..                /* t5 */
  func2 ();
  ..                /* t7 */
}
func1 () {
  ..                /* t2 */
  func2 ();
  ..                /* t4 */
}
func2 () {
  ..                /* t3, t6 */
}
```



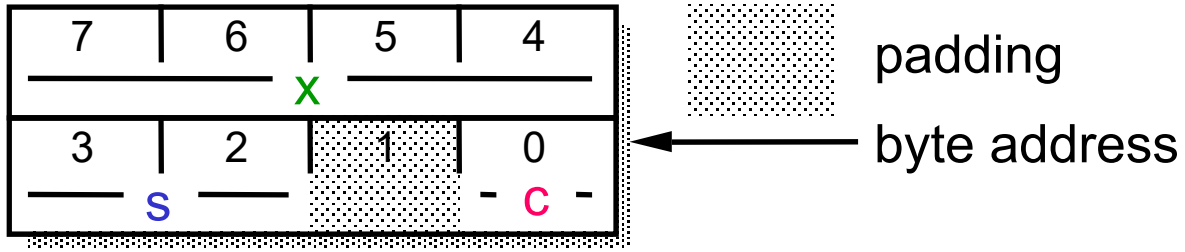
6.9 Use of memory

- Data alignment
  - Where several data items of different types are declared at the same time, the compiler will introduce padding where necessary
  - Struct S1 {char c; int x; short s;} example1;



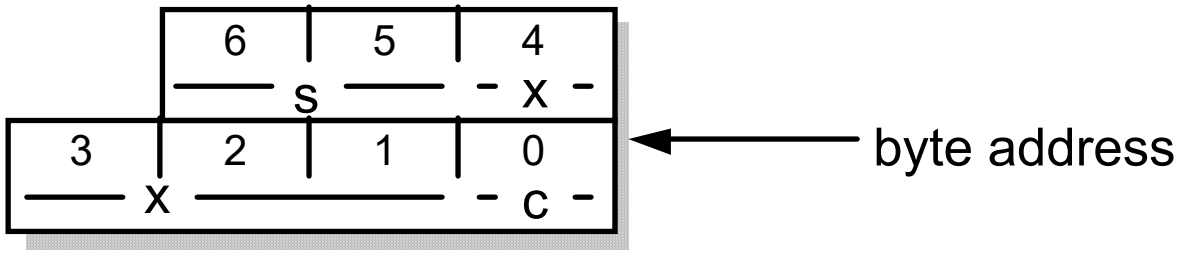
6.9 Use of memory

- Memory efficient
  - The programmer can help the compiler to minimize memory wastage by organizing structures appropriately
  - Struct S2 {char c; short s; int x;} example2;



6.9 Use of memory

- Packed structs
  - The ARM C compiler can produce code that works with packed data structures where all the padding is removed
  - Minimize memory use
  - Incur the overhead of the ARM's relatively inefficient access to non-aligned operands



6.10 Run-time environment

- Minimal run-time library
  - ARM Limited supplies a minimal stand-alone run-time library
  - Once ported to the target environment, allow basic C programs to run
  - It comprise :
    - Division and remainder functions
    - Stack-limit checking functions
    - Stack and heap management
    - Program start up
    - Program termination
  - The total size of the code generated for this minimal library is 736 bytes